A File System Design for the Aeolus Security Platform

by

Francis Peter McKee

S.B., C.S. M.I.T., 2011

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2011

Author.... .....
Department of Electrical Engineering and Computer Science
August 22, 2011

Certified by..... .....
Barbara H. Liskov
Institute Professor
Thesis Supervisor

Accepted by ....
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

A File System Design for the Aeolus Security Platform

by

Francis Peter McKee

Submitted to the Department of Electrical Engineering and Computer Science
on August 22, 2011, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents the design and implementation of a file system for Aeolus, a distributed security platform based on information flow control. An information flow control system regulates the use of sensitive information as it flows through an application. An important part of such a platform is files, since applications use files to store sensitive information. This thesis presents an implementation of a file system that enforces information flow rules on the use of files and generates valuable audit trails of an application's interaction with the file system. My results show that the file system supports information flow control with auditing while performing nearly as well as a native file system.

Thesis Supervisor: Barbara H. Liskov
Title: Institute Professor

Acknowledgments

First and foremost, I want to thank my advisor, Professor Barbara Liskov. I'm very grateful for the energy she invested in my work and the guidance she provided me with while I was here. I feel very fortunate to have worked with someone who, in a relatively short amount of time, I've come to see as a truly great researcher.

I would also like to thank the rest of the Programming Methodology Group, including those visiting from Cornell over the past year. They made me feel welcome during my time here, and the wealth of knowledge they brought to group meetings has sparked my interest in research.

I want to thank the residents of 528 Beacon Street for providing me an outlet in late night video games, food, and adventures. Without all of their distractions, the last four years would have been way too easy and not nearly as fun.

Finally, to all the professors, students, and amazing people I've met at MIT, I leave without a doubt in my mind that I came to the right place.

Contents

Chapter 1

Introduction

This thesis is about the design and implementation of a file system within the context of Aeolus. Aeolus is a platform intended to make it easier to develop secure applications, that is, applications that protect sensitive information entrusted to them. An important part of such a platform is the file system, since files can be used by applications to store sensitive information.

Traditionally files have been protected by access control, but this has proven to be insufficient to prevent information leaks. The problem is that users and applications permitted to access sensitive data may release it accidentally, or even maliciously. Therefore, Aeolus uses a different security model: it tracks information as it flows through the system and allows information to be released only if the user attempting the release is permitted to do so. This way release through accident is prevented, and even some malicious attempts can be thwarted.

The goals for the file system implementation are:

- Ensure absolute safety of information. It must not be possible to circumvent the information flow constraints through the use of files.

- Provide good performance. The goal is to perform approximately as well as the native file system which does not support information flow control. My

implementation meets this goal: experiments show a cost in performance of only 3-6%.

- Provide support for auditing. If information is leaked, administrators need to know how the leak happened and where the information spread. Audit trails also provide a history of events that can give insight into bugs in the system. My implementation provides this information accurately and at low cost.

The remainder of the thesis is structured as follows. Chapter 2 provides relevant background information on the Aeolus platform and how files fit into its security model. Chapter 3 explains the log collection system in Aeolus and describes my approach to auditing the file system and the challenges therein. Chapter 4 explains the user interface to the file system and describes the events used to log its use. Chapter 5 describes the architecture of the system and how its components solve the problems in maintaining security constraints and event logs in the file system. Chapter 6 explains how the event log can be used to analyze a program's interactions with the file system. Chapter 7 evaluates the performance of the file system. Chapter 8 presents some topics for future work and reviews the contributions of the thesis. Appendix A catalogues the events logged by the API and their parameters.

Chapter 2

Files in Aeolus

My file system design is intended to be used as a component of a platform called Aeolus, which was designed to ease the development of secure, distributed applications. In this chapter, I present an overview of the Aeolus security model and a description of the file system in Aeolus. More complete descriptions of Aeolus can be found in Cheng [3] and Popic [6].

2.1 Aeolus Security Model

The Aeolus security model is based on information flow control. It ensures that security constraints on data are enforced on each read and write by a user application. This creates the requirement that programs have the proper authority to perform any attempted operation that bears security concerns. The permissibility of an operation is determined by comparing the contamination of the process attempting the operation with that of the data involved in the operation.

Contamination is captured by means of *labels*. Every entity in the Aeolus system has a secrecy label and an integrity label. These labels are sets of *tags*. Tags provide a way for users to categorize data. In the secrecy label, tags represent types of classificaton. For instance, a user named Bob might have some informa-

tion that he wants to keep secret from his coworkers but share with his family and vice versa. He can capture this constraint by creating the tags BOB_WORK and BOB_FAMILY and putting those tags in the secrecy labels of the appropriate information. In the integrity label, tags represent a type of endorsement on data. For example, Bob may also have some programs on his computer that were installed for him by a trusted friend and others that he downloaded from a questionable website. To protect the integrity of the files on his computer, he could create a tag, TRUSTED, to go in the integrity label of the files that can only be written to by his friends' programs, to indicate that they are not corrupt.

In order for information to flow between two entities, as in a process reading some data from a file, the following constraints are imposed on the information and receiver by the system.

$$\begin{aligned} \text{SECRECY}_{\text{information}} &\subseteq \text{SECRECY}_{\text{receiver}} \\ \text{INTEGRITY}_{\text{information}} &\supseteq \text{INTEGRITY}_{\text{receiver}} \end{aligned}$$

These constraints ensure classified data remains secret. For example, if a file has the label {CONFIDENTIAL}, a process's secrecy label must contain the tag CONFIDENTIAL in order to read the file. The integrity of endorsed data is also protected. For example, a file with the integrity label {ADMIN}, can only be modified by a process whose integrity label contains the ADMIN tag.

2.2 Information Flow in the File System

Aeolus allows the labels of processes to vary over time, but the labels of data objects cannot change. Thus, the labels of files are immutable. A security and in-

tegrity label must be assigned to each file at the time of creation, and they cannot be changed for the life of the file.

The following constraints are also imposed on the directory tree structure of the file system in order to prevent information flow violations through file system operations.

$$\begin{aligned} \text{SECRECY}_{\text{directory}} &\subseteq \text{SECRECY}_{\text{child}} \\ \text{INTEGRITY}_{\text{directory}} &\supseteq \text{INTEGRITY}_{\text{child}} \end{aligned}$$

The secrecy constraint is based on the contamination that processes incur from reading a path name. In order to read or write a file in a particular location, a process must read each of the directories in the path. This means the process is at least as contaminated as the parent directory. Therefore, if the child were not also at least as contaminated as the directory, no process could write to it, and the constraint on reads would be meaningless, because any process that could determine the file existed would already be contaminated enough to read it.

The integrity constraint exists due to the fact that operations performed on a directory can affect the children of that directory. For example, if a high integrity file existed in a low integrity directory, a process with only the integrity of the directory could delete the directory, removing the file along with it, and then replace it with a file of the same name but lower integrity. It is clearly a security violation for a low integrity process to delete a high integrity file. Thus, we constrain files to have at least the integrity of their parent directories in order to prevent that situation.

For information flow purposes the root directory of an Aeolus file system is a special case. The secrecy label of the root directory is always empty. If this were

not the case, every process would have to have at least the secrecy of the root directory to use the file system at all. This would either make that particular level of secrecy meaningless or it would render the file system unusable. The integrity label of the root directory is complete, that is, it contains all tags. This is required by the constraints on the directory structure. Applying the standard rules of information flow, a process should then need complete integrity in order to write to the root directory. This, however, is impractical. A process with complete integrity would execute with a prohibitively high amount of privilege. For this reason, a special write constraint is applied to the root directory. A process may write to the root directory regardless of its integrity label, but files that are created in the root directory cannot be given any more integrity than the process that created the file. Files also cannot be removed from the root directory by a process with less integrity than the file. These constraints preserve the semantics of integrity while maintaining the usability of the file system.

2.3 Aeolus System Model

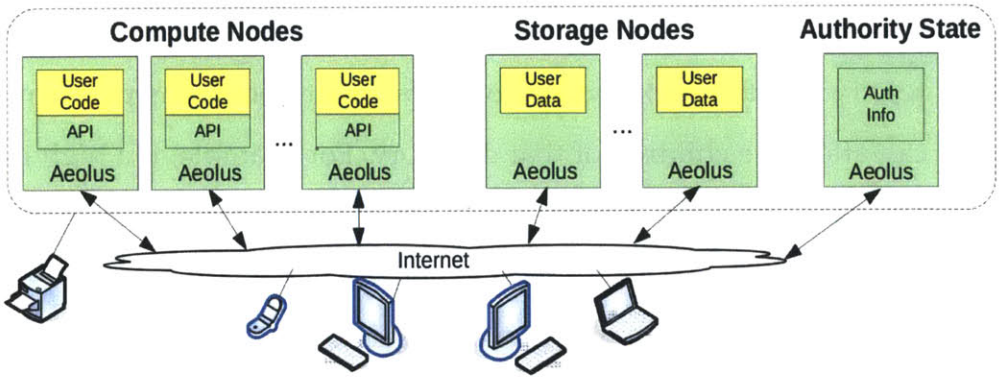

Figure 2-1: The high level design of the Aeolus Distributed Security Platform

Individual machines in an Aeolus network are referred to as nodes. Each node can also run multiple virtual nodes. The set of nodes contains both compute nodes and storage nodes as illustrated in Figure 2-1. Compute nodes run user applica-

tions which interact with a trusted Aeolus system process in order to access system resources (such as the file system). Storage nodes run an Aeolus system process that moderates I/O to persistently stored user data on the machines on which they are running. While Figure 2-1 shows compute nodes and storage nodes in different locations, nothing prevents a storage node and a compute node from running on the same machine. In either case, application code uses the DNS host name to reference the location of a particular storage node. All of these nodes access a shared central authority server in order to determine if the proper permissions exist for any attempted operations.

Chapter 3

Auditing Strategy

Creating audit trails that record the use of a system is a critical part of security. If information is leaked, administrators need to know how the leak happened and where the information spread. Audit trails also provide a history of events that can give insight into bugs in the system, including those that are not security related. In this chapter, I present an overview of the auditing system in Aeolus and describe the design choices in auditing the file system.

3.1 Log Collection in Aeolus

Every call to a runtime procedure in the Aeolus API generates an event in the log. Each node in an Aeolus network is responsible for generating events for the operations that occur at that node. The authority server also generates events for authority operations such as tag creation or authority delegation. These logs are batched and shipped to a node that is designated for log collection. There the events are processed and stored in a database. Figure 3-1 illustrates this process.

Aeolus establishes an ordering of events that reflects causality by attaching information to each event about that event's predecessors in the program execution. Each event is associated with the last event generated by the process in which the

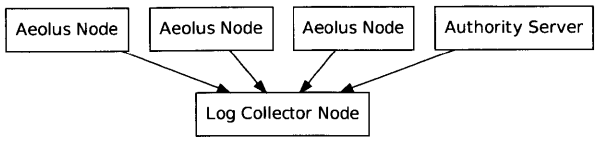

Figure 3-1: Flow of logs to the log collection system.

event happened. It may also be associated with additional causal predecessors. For example, if a client node issues an RPC to a server node, two events are generated on each node. An event is generated at the client when the RPC is issued, and an event is generated at the server when the RPC is received. After the procedure is executed an event is generated at the server when a response is sent. A final event is generated at the client when the response is received. This final event has two predecessors. It is preceded by the event for the RPC being issued. It also has a causal predecessor of the event generated at the server when the response was sent. The correlation of events in this process is illustrated in Figure 3-2.

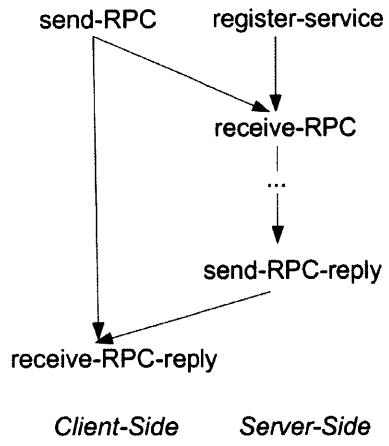

Figure 3-2: Remote Procedure Call Events

A complete description of the current implementation of event auditing in Aeolus can be found in Popic [6] and Blanks [1].

3.2 Logging Strategy in the File System

The goal of the logging strategy in the file system is to maintain event causality as accurately as possible, in balance with other platform requirements such as efficiency. In order to achieve this, it is important to consider exactly what the notion of causality means in the context of the file system. Since the purpose of Aeolus is regulating the flow of information, following the chain of causal links between events would ideally parallel the flow of information through the system. In the file system this means that the causal predecessor of any read event should be the write event that created the information being read. If this were known, an examiner of the log would be able to tell exactly how information flowed between processes reading and writing to the file system. This knowledge would be valuable in the event of an information leak or for general debugging.

There are several approaches to achieving this kind of causality to varying degrees of accuracy and with varying performance. One possible design is to interpose Aeolus code in every use of files by a user application. This would make it possible to record information about every read and write event as it happened so that the predecessor event could be provided accurately for each.

One approach to interposition is to implement client side caching at the platform level. For this approach to get performance close to that of a native file system, however, it requires re-implementing all of the file caching machinery that already exists at the operating system level, substantially increasing the complexity of the implementation. An earlier design for the file system described and implemented in Popic [6] used this approach. This achieved the type of causality described above in systems that practice concurrency control. It only tracked the flow of information at the granularity of files, so it did not ensure that pages written by a causal predecessor were actually seen by a read, but such functionality could have been implemented at the platform user level. However, this imple-

mentation did not have efficient caching, so it was slow.

Another approach to interposition is to modify the operating system to log and track predecessors as it does the cache management for files. This approach would be efficient, and it would allow the platform to support page level logging of file access. Since the operating system manages files at the page level, it could keep a record of the last write for each cached block. However, this would require modifying operating system code, which would make Aeolus platform dependent and would increase the difficulty of code maintenance.

This thesis investigates an alternative strategy to interposition. The goal is to avoid interfering with most of what happens in the file system at both the server and client. Information about causality is obtained by comparing timestamps taken on each read or write. The timestamps are taken by checking the last modified time of the file, which is set by the server's file system every time a page of the file is written.

If files are used without conflicting reads and writes running concurrently, this strategy can provide an accurate ordering of events, e.g. the time logged when a read stream is opened will be the time logged of the last write to that file. However, there are scenarios in which using timestamps cannot provide a perfect ordering of events; this is discussed further in Chapter 6.

Chapter 4

File System Interface

The Aeolus platform is designed to provide abstractions that help programmers more easily develop secure, distributed applications. These abstractions are targeted at developers familiar with Java. Maintaining this pattern, the operations provided by the file system and the interface displayed to the user are based on several classes from the java.io package of the Java Platform, Standard Edition 6 [5].

The rest of this chapter describes the operations provided by the file system interface and the events generated by each operation. The operations are organized by the class to which they belong in the Aeolus File API.

4.1 AeolusFile

The AeolusFile class is modeled after java.io.File. It includes a subset of the methods provided by its Java counterpart. The method to set the last modified time of the file is intentionally excluded, because Aeolus uses the last modified time to generate timestamps as described in Chapter 5. Other methods, mostly concerned with reading the attributes of the file could have been included if they were treated as reading the file for information flow purposes. This implementation includes a

minimal subset that provides necessary functionalities. The remainder of this section describes those methods.

4.1.1 AeolusFile(String path, String host)

The constructor for the AeolusFile object takes a string that specifies the path name of the file and a string that specifies the DNS host name of the storage node where the file is located. These values are stored and used as implicit parameters to the other methods of AeolusFile. No events are logged by the constructor.

4.1.2 boolean createNewFile(Label secrecy, Label integrity)

This method creates a new file if one does not already exist at the path specified. Its arguments are the secrecy and integrity labels that the caller wishes to give to the new file. The caller must have equivalent secrecy and integrity labels to the parent directory in order for the operation to be permissible. The method returns true if the file was created.

The operation creates two events at the node on which it is called and one at the host where the new file is to be located. When the call is issued, the caller logs a `CREATE_FILE` event. When the host executes the operation it logs an `FS_CREATE_FILE` event. This event has predecessors of the `CREATE_FILE` on the caller and the event of the last modification of the parent directory on the host. Finally, when the caller receives a response from the host, it logs a `CREATE_FILE_REPLY` event with predecessors of both the `CREATE_FILE` event and the `FS_CREATE_FILE` event. All of these events are annotated with the path and host name of the created file. Figure 4-1 illustrates the correlation between these events.

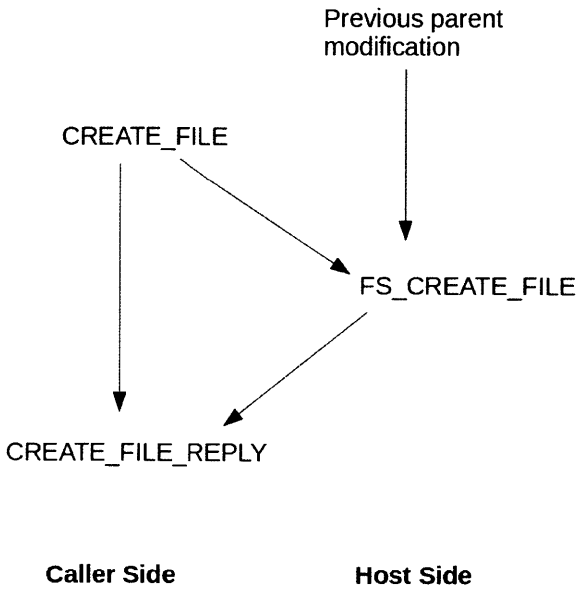

Figure 4-1: File Creation Events

4.1.3 boolean mkDir(Label secrecy, Label integrity)

This method creates a new directory at the specified path as long as no other file exists at that location. Its arguments are the secrecy and integrity labels that the caller wishes to give to the new directory. The caller must have equivalent secrecy and integrity labels to the parent directory in order for the operation to be permissible. The method returns true if the directory was created.

The operation creates two events at the node on which it is called and one at the host where the new directory is to be located. When the call is issued, the caller logs a `CREATE_DIRECTORY` event. When the host executes the operation it logs an `FS_CREATE_DIRECTORY` event. This event has predecessors of the `CREATE_DIRECTORY` on the caller and the event of the last modification of the parent directory on the host. Finally, when the caller receives a response from the host, it creates a `CREATE_DIRECTORY_REPLY` event with predecessors of both the `CREATE_DIRECTORY` event and the `FS_CREATE_DIRECTORY` event. All of these events are annotated with the path and host name of the created directory. Fig-

ure 4-2 illustrates the correlation between these events.

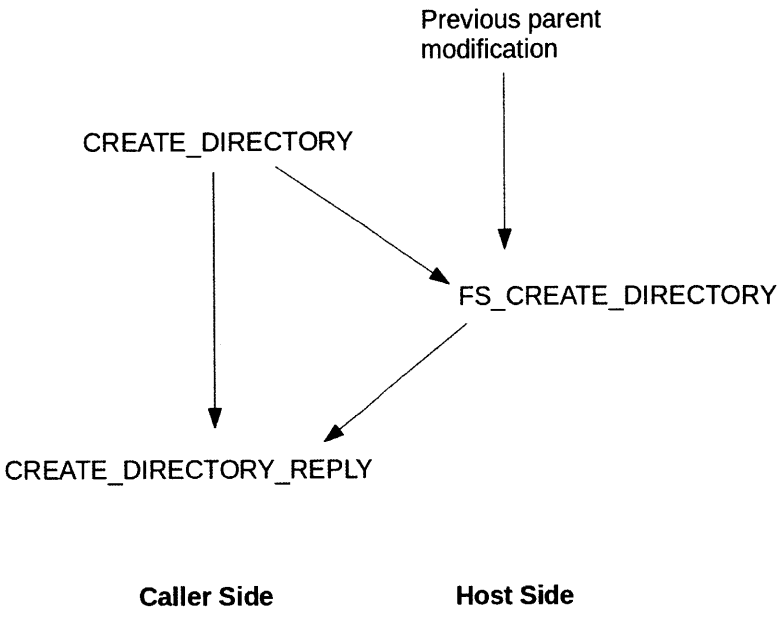

Figure 4-2: Directory Creation Events

4.1.4 boolean delete()

This method deletes any file or empty directory located at the specified path. It requires no other arguments than the path and host provided by the AeolusFile object. The caller must have equivalent secrecy and integrity labels to the parent directory in order for the operation to be permissible. The method returns true if a file was deleted.

The operation creates two events at the node on which it is called and one at the host from which the file is to be removed. When the call is issued, the caller logs a `DELETE` event. When the host executes the operation it logs an `FS_DELETE` event. This event has predecessors of the `DELETE` on the caller and the event of the last modification of the parent directory on the host. Finally, when the caller receives a response from the host, it creates a `DELETE_REPLY` event with predecessors of both the `DELETE` event and the `FS_DELETE` event. All of these events

are annotated with the path and host name of the deleted file. Figure 4-3 illustrates the correlation between these events.

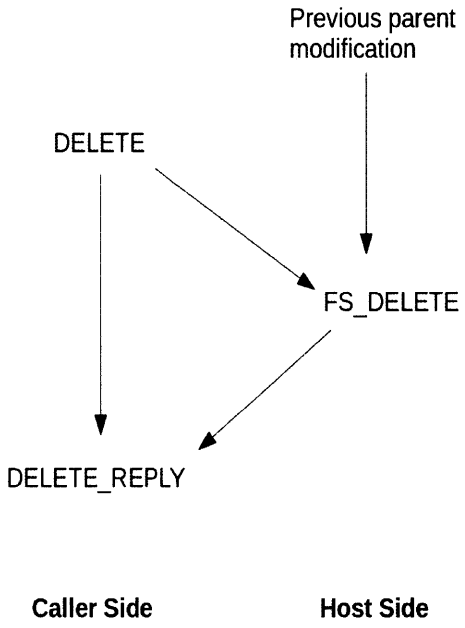

Figure 4-3: Delete Events

4.1.5 Label getSecrecy()

This method returns the secrecy label of the file at the specified path, if one exists. It takes no other arguments than the path and host provided by the AeolusFile object. The caller's secrecy label must contain that of the parent directory in order for the operation to be permissible.

The operation logs a single `GET_SECRECY_LABEL` event at the node where the method is called. This event is annotated with the path and host name of the file, as well as the last modified time of the parent directory. The information returned reflects the state of the directory as of that time.

4.1.6 Label getIntegrity()

This method returns the integrity label of the file at the specified path, if one exists. It takes no other arguments than the path and host provided by the AeolusFile object. The caller's secrecy label must contain that of the parent directory in order for the operation to be permissible.

The operation creates a single `GET_INTEGRITY_LABEL` event at the node where the method is called. This event is annotated with the path and host name of the file, as well as the last modified time of the parent directory. The information returned reflects the state of the directory as of that time.

4.1.7 String[] list()

This method returns an array of the names of the files in the directory located at the specified path. If the file at the specified path is not a directory, the return value is null. The caller's secrecy label must contain that of the directory in order for the operation to be permissible.

The operation creates a single `LIST_DIRECTORY` event at the node where the method is called. This event is annotated with the path and host name of the directory, as well as the last modified time of the directory. The information returned reflects the state of the directory as of that time.

4.2 AeolusFileOutputStream

The AeolusFileOutputStream is modeled after java.io.FileOutputStream. It allows users to modify files by exposing the write operation. It also logs events for the open and close of the stream but not for each write. This is consistent with the close to open semantics of the NFS protocol underlying this implementation of the Aeolus File System. This section describes the methods of the AeolusFileOutput-

Stream.

4.2.1 AeolusFileOutputStream(String path, String host, boolean append)

The constructor takes the path and host name of the target file, as well as a boolean indicating whether the stream should write to the beginning or the end of the file. An `OPEN_WRITE_FILESTREAM` event is generated at the node where a stream is opened. This event is annotated with the path and host name of the file as well as the last modified time of the file at the time it was opened for writing.

4.2.2 void write(byte[] buffer)

The write operation takes a buffer of data and writes it to the file. This operation is only permissible if the caller's secrecy and integrity labels are equivalent to those of the file. If the caller does not have the correct labels, an `FS_WRONG_LABELS` event is logged and the stream is closed. The `FS_WRONG_LABELS` event is annotated with the path and host name of the file. If the write is successful, no events are logged for this operation.

4.2.3 void close()

The close operation flushes any data remaining in the stream to the file and closes the stream. When the stream is closed a `CLOSE_FILESTREAM` event is created. Its predecessor is the event for the last operation in the process. It also contains the last modified time of the file after the stream was closed.

4.3 AeolusFileInputStream

The AeolusFileInputStream is modeled after java.io.FileInputStream. It exposes the read method. It also employs the same logging semantics as the AeolusFileOutputStream.

4.3.1 AeolusFileInputStream(String path, String host)

The constructor takes the path and host name of the target file as parameters. An `OPEN_READ_FILESTREAM` event is generated at the node where a stream is opened. This event is annotated with the path and host name of the files as well as the last modified time of the file at the time it was opened for reading.

4.3.2 int read(byte[] buffer)

The read operation takes a buffer and fills it with data from the file until the buffer is full or the end of the file is reached. This operation is only permissible if the caller's secrecy label contains that of the file. If the caller does not have the correct labels, an `FS_WRONG_LABELS` event is logged and the stream is closed. The `FS_WRONG_LABELS` event is annotated with the path and host name of the file. If the write is successful, no events are logged for this operation.

4.3.3 void close()

The close operation closes the stream. When the stream is closed a `CLOSE_FILESTREAM` event is created. Its predecessor is the event for the last operation in the process. It also contains the last modified time of the file at the time the stream was closed.

Chapter 5

Implementation

The Aeolus file system layer is implemented in approximately 2000 lines of Java code. It supports the NFS [4] file system. It requires that each storage node's NFS file server is mounted in a directory that is provided to Aeolus as a configuration parameter. It also requires that Aeolus is run with a system principal that cannot be used by other users. Aeolus will only make files readable and writable by this principal. Furthermore, traffic between Aeolus nodes needs to be encrypted. This can be done using secure shell tunneling or a virtual private network. These configurations are critical to prevent information leaks.

5.1 Architecture

All Aeolus nodes are compute nodes that can run user application code. Some nodes can also be storage nodes. Each storage node must host an NFS file server. If multiple storage nodes exist in an Aeolus deployment, there will be multiple file systems.

In order to access a particular file system, application code must identify it. This is done by using the DNS host name of the machine where the file system resides. This name is used to open a read or write stream, or to create an AeolusFile object.

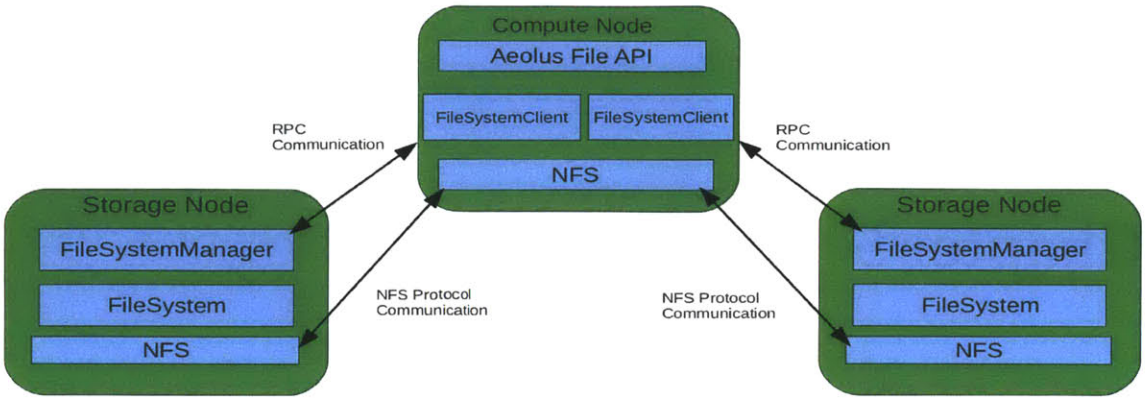

Figure 5-1: An overview of the architecture of the Aeolus file system layer

Each storage node runs the NFS file server code, as well as an Aeolus file system `manager`. The `manager` is used to handle file creates and deletes as described later in this chapter. It uses NFS to do the actual I/O.

Each compute node runs an Aeolus file system `client` for each storage node that it accesses. Use of files in a file system by application code on a compute node is handled by the `client` on that node for that file system.

Figure 5-1 illustrates the file system architecture in an example network with one compute node that accesses data on two storage nodes.

5.2 Label Storage

The Aeolus file system requires that every file have associated secrecy and integrity labels. As mentioned, these labels are created when the file is created and are immutable. Furthermore, all accesses to a file are mediated through the labels as discussed in Chapter 2.

These semantics require that labels be stored somewhere on disk, separate from the files themselves. In this design the labels are stored in a hidden metadata file associated with the parent directory of the file. This is more efficient in time and

space than the design in Popic [6], which used a separate label storage file for each file. The implementation in Cheng [3] stored the labels in a relational database. This design is effective but complicates deployment of the Aeolus platform. Using one metadata file per directory requires an extra write operation each time a file is created, but it relieves the system from dependence on an external database.

5.3 Accurate Information Flow

Aeolus requires that there be no errors in file access that could undermine information flow control. In particular, whenever a program reads or writes a file, its labels and those of the file must allow the actions. However, since information about labels is stored separately from the file, effects from client side caching create potential for error. For example, if one client deleted a file and then created another with the same name but different labels immediately after, a different client might see stale label data that would allow it access that it should not have to the new file.

The solution in this design is to avoid such problems using a renaming scheme that assigns each file a unique identifier, independent of the name given to it by the creator. Directories use this identifier to reference files. In this way, every incarnation of a file is uniquely identified and there can be no confusion about the labels of a particular incarnation.

The unique identifier of a file is stored in the same directory metadata file that stores the file's labels. The format for each line in the metadata file is shown below.

```
<file_path_name>|<file_uid>|<secrecy_label>|<integrity_label>
```

This solution requires that following a path name not only means reading all of the directories in the path but also their associated metadata files. This is done step

by step to find the unique path representing a path requested by an application. This doubles the number of reads for some file accesses, but it can be made efficient using caching as described in Section 5.5.

5.4 File System Manager

The file system `manager` handles all of the storage node activities that are not directly related to putting data on disk.

The `manager` provides a remote interface to the file system which is used by each of the `clients` accessing data on that storage node. It keeps a registry of `clients`, and it processes RPC's received from them.

The `manager` receives requests from clients to create or delete files. The information from each of those requests is forwarded to the file system and stored for event logging purposes. Since all operations that modify a directory are processed by the `manager`, it is capable of maintaining a table of the most recent modifications of each directory. This allows it to create events for each of the operations and attach the correct predecessors as described in Chapter 4.

5.5 File System Client

The file system `client` is the local interface between each compute node and each storage node that it accesses. Each compute node has a separate `client` for each storage node. All calls to file system operations provided by the Aeolus runtime are made through a `client`.

5.5.1 Communication Strategy

This design for the Aeolus file system aims to leverage the speed and efficiency of a native NFS client as much as possible. However, NFS provides weak concur-

rency protections that could lead to security constraint violations in an environment where many compute nodes are creating and deleting files.

To protect against this possibility, the `client` is designed with the principle that all operations that alter the directory structure on disk should be carried out on the machine where the disk resides. Any create or delete operations are executed by issuing an RPC to the `manager` on the desired host using the Java Remote Method Invocation system.

Operations that simply read or write an existing file are executed through NFS and utilize the operating system level cache.

5.5.2 Observing Metadata

Each of the `client` operations that reads or writes a file through NFS follows the same protocol to find the file and ensure that constraints on labels are observed. When the `client` receives a path name for a file from a user for the first time, it must determine the actual unique path of the file on disk. It does this by reading metadata files for each parent in the path, starting with the closest one for which the unique identifier is already known. In the worst case, this will be the root. This process can be interrupted at any point if a node in the path cannot be read with the labels of the process. Each read of a metadata file must be atomic, that is, there must not be a concurrent write to the file. The `client` ensures this by taking the last modified time of the metadata file when it is opened for reading and comparing it to the last modified time after the read has finished. If the two do not match, the operation is repeated. Failure to ensure this read is atomic could result in the path or label data being compromised. When the unique identifier for a file is determined, it is cached in an in-memory table that maps from a user path name to an object that stores the unique path name and the file's labels. Figure 5-2 shows the potential entries to the table generated by looking up the path "/foo/bar". The entries in this table are deleted or updated when a unique path found in the table

cannot be found on disk, indicating that a file has been deleted or reincarnated.

"/foo"	["/aeolusroot/347", {SECRET}, {TRUSTED}]
"/foo/bar"	["/aeolusroot/347/842", {SECRET, TS}, {}]

Figure 5-2: Example table entries generated from looking up the path "/foo/bar"

5.5.3 AeolusFile Operations

The AeolusFile operations that modify a directory (createNewFile, mkDir, and delete) are implemented on the `client` as RPC's made to the `manager`. The remaining AeolusFile operations (getSecrecy, getIntegrity, and list) only require reading a directory metadata file and are implemented in the manner described in the previous section.

5.5.4 Stream Operations

The open operation of Aeolus streams looks up the labels of the target file to ensure the open is permissible, then it opens the underlying Java file stream. The read and write operations check the calling process's labels to ensure use of the stream is still allowed, then either carry out the operation or close the stream, depending on the result of the check. The close operation simply flushes the underlying stream and frees the resources associated with it.

5.5.5 Logging

The `client` is responsible for retrieving the timestamps that are used in the file system auditing strategy described in Chapter 3. The `client` gets timestamps using the `last_modified` attribute of the file involved in the operation being

logged. The `last_modified` attribute is a Unix timestamp set by the native file system when the contents of a file are modified. It is set using the system clock of the storage node, providing a consistent time reference for comparing the ordering of events. It is also used in the cache coherency protocol of NFS to determine when cached file contents need to be invalidated because contents on disk have changed. This means that the Aeolus file system can efficiently access the timestamp when it does so in conjunction with a read or write to a file, because the NFS client fetches it regardless of whether or not Aeolus accesses it. The last modified time is retrieved once per operation that the `client` performs.

Chapter 6

Analyzing Audit Trails

All of the events described in Chapter 4 are sent to the log collection and storage system described in Popic [6] and Blanks [1]. They are eventually stored as rows in an event table in a PostgreSQL [7] database. There they can be queried to learn about the access history of the file system.

In the rest of this chapter I describe how to construct file access history from the event log and exactly what can be known about a file's history from the information in the log.

6.1 Querying the Event Log

The event table in the database used by the Aeolus log storage system has 36 columns, many of which are only used for particular events. The following two columns are used in the log entries for all runtime operations.

- `op_name` : a number that represents the type of event
- `process` : the id number of the process from which the operation was called

These next columns are used just in logging file system events.

- `filename` : a string indicating the path name of the file used in an operation

- `hostname` : a string indicating the DNS name of the machine the file is on
- `fs_timestamp` : a timestamp retrieved from the last modified attribute of a file during certain operations

One of the most common types of queries related to the file system is likely to be an examination of the access history of a particular file. The following is an example query that would return each of the events related to a file at the path, "/test," on a host named "web.mit.edu" in chronological order of their completion.

```
SELECT * FROM EVENTS
 WHERE filename = '/test' AND hostname = 'web.mit.edu'
 ORDER BY fs_timestamp;
```

Another potentially common type of query is one that seeks to find all interaction with the file system originating from a particular process (e.g. in order to track what a process may have written to disk as its contamination changed over time). Below is an example of such a query.

```
SELECT * FROM EVENTS
 WHERE process = 1 AND filename IS NOT NULL
 ORDER BY fs_timestamp;
```

6.2 Concurrency Control

An important notion in analyzing an application is whether or not concurrency control is practiced when using the file system. The event log can indicate whether or not file access is properly controlled, and knowing this has an impact on what knowledge about information transfer can be gained from reviewing the event log. I define file access in an Aeolus file system as practicing concurrency control if each of the following two constraints on the event log are met.

1. For every pair of read stream events whose open has `fs_timestamp`, t_0, and whose close has `fs_timestamp`, t_1, $t_0 = t_1$

2. For any two write stream event pairs with open `fs_timestamp` values of a_0 and b_0 and close `fs_timestamp` values of a_1 and b_1, respectively, either $a_0 < a_1 \leq b_0$ or $b_1 \leq a_0 < a_1$

These constraints means that a file has the same contents on disk throughout the course of any read, and there is never any more than one write to a particular file executing at once.

If these constraints are met, then the guarantees provided by the event log are strong and simple. In a chronological ordering of events affecting a particular file, such as that which would be generated by the example query in the previous section, a read will see the aggregate effects of all the writes before it on the list. Any writes that come after a read cannot affect what data was seen by the read.

Of course, Aeolus only logs that a read or write happened, not which pages were actually involved in the operation. This means that it is not clear that information was actually transferred between any read and write operations. It is only clear that such a transfer could have happened. Application level logging could be used to gain more certainty.

If the constraints are not met, then the guarantees about file access are weaker. It is still the case that any read that begins after a write terminates will not miss the effects of that write. If, however, a read begins before a write terminates or a write begins in the middle of a read, the data seen by the read may reflect some, all, or none of the modifications made by the write.

Chapter 7

Performance Evaluation

In this chapter I present and analyze the results of an experiment to evaluate the performance of the file system. I performed this experiment using two computers with 4 GB of physical memory, 7200 RPM hard drives, and Intel Q9550 CPU's which have four cores and a clock speed of 2.83 GHz. The computers were connected by a 1 Gigabit LAN connection with a latency of approximately 0.5 milliseconds. The Aeolus platform and application was built for the 64-bit OpenJDK JVM (build 19.0-b09) and utilized NFS version 4 on the 2.6.31-23 GNU/Linux kernel.

7.1 Experiment Design

This experiment involved running the Aeolus runtime on two machine. One of the machines hosted an NFS server and was running Aeolus only for the purpose of providing the File System Manager's remote interface to the other machine. The other machine ran an Aeolus application implemented for this experiment.

The application was a benchmark based on the widely used file system benchmark, Bonnie [2]. Bonnie is implemented in C which provides a more fine grained interface to I/O than either Java or the Aeolus File API, so the version implemented for this experiment varies slightly in low level details from Bonnie. A high level

description of the benchmark follows.

1. Create a new file on the storage node machine.

2. Write 10 MB to the file, a single byte at a time.

3. Read and then write over each chunk (16 KB) in the file

4. Write over all 10 MB of the file, one chunk at a time

5. Read all 10 MB of the file, a single byte at a time.

6. Read all 10 MB of the file, one chunk at a time

This application was run on the second machine which had an NFS client and had mounted the file system on the first machine.

As a control, the same benchmark was implemented as a standard Java application. This version performed the exact same operations but used the corresponding java.io classes rather than the Aeolus File API. The control application did not interact with the Aeolus platform at all and no Aeolus processes were active on either machine while it executed. Instead, it performed all operations on a file that it created in the directory on the client machine where the NFS server of the storage machine was mounted.

Following the execution of both the Aeolus and Java benchmarks, the entire experiment was repeated with the benchmarks run from the machine where the NFS server was located. In this second set up, both the file system and the application were located on the same machine, so the I/O operations were performed locally rather than across a network connection. This was done to control for the possibility that network communication times dominated the measurements, and that an application run from a storage node would have relatively worse performance.

Test	**Aeolus File System** (time in ms)	**Java.io on NFS** (time in ms)
File Create	25	59
Byte-by-Byte Write	15625	15092
Chunk-by-Chunk Read then Write	1121	1118
Chunk-by-Chunk Write	1111	1111
Byte-by-Byte Read	6295	5908
Chunk-by-Chunk Read	8	8
Total Running Time	**24185**	**23296**

Table 7.1: Execution times for the file system benchmark run across the network

Test	**Aeolus File System** (time in ms)	**Java.io on NFS** (time in ms)
File Create	17	0
Byte-by-Byte Write	23205	22244
Chunk-by-Chunk Read then Write	24	24
Chunk-by-Chunk Write	146	140
Byte-by-Byte Read	6174	5679
Chunk-by-Chunk Read	5	4
Total Running Time	**29568**	**28091**

Table 7.2: Execution times for the file system benchmark run locally

7.2 Results

The benchmark implemented in Aeolus and the control were run 22 times each and execution times were recorded for each step in the benchmark for the last 20 iterations. The unrecorded iterations were intended to eliminate influence on the data from any transient initial effects. The execution times averaged over the 20 recorded iterations across the network are displayed in Table 7.1. The measurements for the experiment performed on one machine are shown in Table 7.2. The standard deviations of the total running times for both the Aeolus and the Java tests were less than 1% of the averages.

7.3 Analysis

The results show that the Aeolus File System is 3.8% slower on this benchmark than java.io on NFS when run across the network and 5.3% slower when the benchmark is run locally. Counter-intuitively, the benchmark took longer when run locally by both systems. This is because the NFS client is willing to batch sets of small writes before sending them across the network and writing them to disk, whereas writing to the disk from the server involves more frequent synchronous I/O. When running either across the network or locally, though, the relative performance penalty from Aeolus is a small price to pay for the information flow guarantees provided by the Aeolus File System.

One point to note from the data is that the performance of the two systems is nearly identical on the tests that wrote or read entire chunks instead of individual bytes. This is because the costs of the Aeolus File System scale with the number of I/O operations but not with the amount of data that is read or written. This shows that an application which was able to do its I/O in large batches rather than many smaller operations would be able to get performance very close to using standard Java.

Another point of interest is that the Aeolus File System outperformed the control on file creation time across the network. This is due to the way file creates work in NFS. When an NFS client creates a new file, several additional network communications happen with the server in order to exchange a file handle and other metadata that make writing to the new file more efficient. In the Aeolus File System, the file is not created on the client. Instead, a Java RPC is sent requesting that the server create a new file. This means that the extra information needed for the client to write to the new file is not exchanged until the client first attempts to write to the file. Therefore, the improved performance on the file creation time corresponds with a decrease in performance in the subsequent test of writing to

the file.

Chapter 8

Future Work and Conclusions

This chapter summarizes the contributions of this thesis and presents some possible future work on the Aeolus file system.

This thesis has presented the design and implementation of a file system for the Aeolus security platform. I presented a strategy for logging events in the system and a means for using the event log to analyze the interactions of an application with the file system.

This thesis makes the following contributions:

1. I present an effective strategy for logging in the file system
2. I provide a technique for determining event causality and an application's adherence to concurrency control from the event log
3. I implemented and tested a complete file system layer for the Aeolus security platform. This system maintains security constraints and event logs.

One of the most important next steps for the Aeolus platform and the file system is developing a plan for recovery. In the case of a storage node failure, contents on disk could be left inconsistent in a way that might allow security constraints to be violated. A strategy for dealing with the fact that writing files and labels is not atomic could solve this.

Bibliography

[1] Aaron Blankstein. Analyzing audit trails in the Aeolus security platform. Master's thesis, MIT, Cambridge, MA, USA, June 2011.

[2] Tim Bray. *Bonnie File System Benchmark.* Available from: `http://www.textuality.com/bonnie`.

[3] Winnie Wing-Yee Cheng. *Information Flow for Secure Distributed Applications.* Ph.D., MIT, Cambridge, MA, USA, August 2009. Also as Technical Report MIT-CSAIL-TR-2009-040.

[4] Internet Engineering Task Force. *Network File System Version 4 Protocol.* The Internet Society, 2003. Available from: `http://www.ietf.org/rfc/rfc3530.txt`.

[5] Oracle. *Java Platform, Standard Edition 6,* 2011. Available from: `http://download.oracle.com/javase/6/docs/api`.

[6] Victoria Popic. Audit trails in the Aeolus distributed security platform. Master's thesis, MIT, Cambridge, MA, USA, September 2010. Also as Technical Report MIT-CSAIL-TR-2010-048.

[7] PostgreSQL Development Team. *PostgreSQL 9.0.4.* PostgreSQL Global Development Group, 2010. Available from: `http://www.postgresql.org`.

Appendix A

Audit Trail Events

This appendix catalogues the events that are generated by each call to the Aeolus File API and lists their attributes. Certain attributes are common to all events. These include the process identifier, the virtual node identifier, the time logged, and other metadata that applies to any event. These common attributes are not included in this listing.

A.1 AeolusFile

Method Events
bool createNewFile(secrecy, integrity)
CREATE_FILE:
params= {hostname, filepath, secrecy, integrity} preds ={last_event_in_thread}
FS_CREATE_FILE:
params= {hostname, filepath} preds = {create_file, last_parent_mod}
CREATE_FILE_REPLY:
params= {hostname, filepath} preds = {fs_create_file, create_file}

Method Events
bool mkDir(secrecy, integrity) CREATE_DIRECTORY: params= {hostname, filepath, secrecy, integrity} preds ={last_event_in_thread} FS_CREATE_DIRECTORY: params= {hostname, filepath} preds = {create_directory, last_parent_mod} CREATE_DIRECTORY_REPLY: params= {hostname, filepath} preds = {fs_create_directory, create_directory}
bool delete() DELETE: params= {hostname, filepath} preds ={last_event_in_thread} FS_DELETE: params= {hostname, filepath} preds = {delete, last_parent_mod} DELETE_REPLY: params= {hostname, filepath} preds = {fs_delete, delete}
label getSecrecy() GET_SECRECY_LABEL: params= {hostname, filepath, fs_timestamp} preds= {last_event_in_thread}
label getIntegrity() GET_INTEGRITY_LABEL: params= {hostname, filepath, fs_timestamp} preds= {last_event_in_thread}
string[] list() LIST_DIRECTORY: params= {hostname, filepath, fs_timestamp} preds= {last_event_in_thread}

A.2 AeolusFileOutputStream

Method Events
AeolusFileOutputStream(hostname, filepath) `OPEN_WRITE_STREAM:` params= {hostname, filepath, fs_timestamp} preds= {last_event_in_thread}
void write() `FS_WRONG_LABELS:` params= {hostname, filepath} preds= {last_event_in_thread}
void close() `CLOSE_FILESTREAM:` params= {hostname, filepath, fs_timestamp} preds= {last_event_in_thread}

A.3 AeolusFileInputStream

Method Events
AeolusFileOutputStream(hostname, filepath) `OPEN_WRITE_STREAM:` params= {hostname, filepath, fs_timestamp} preds= {last_event_in_thread}
void write() `FS_WRONG_LABELS:` params= {hostname, filepath} preds= {last_event_in_thread}
void close() `CLOSE_FILESTREAM:` params= {hostname, filepath, fs_timestamp} preds= {last_event_in_thread}